BERMUDA - AN ARCHITECTURAL PERSPECTIVE ON INTERFACING PROLOG TO A DATABASE MACHINE

by

Yannis E. Ioannidis
Joanna Chen
Mark A. Friedman
Manolis M. Tsangaris

Computer Sciences Technical Report #723

October 1987

BERMUDA — AN ARCHITECTURAL PERSPECTIVE ON INTERFACING PROLOG TO A DATABASE MACHINE

Yannis E. Ioannidis
Joanna Chen
Mark A. Friedman
Manolis M. Tsangaris

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

(Submitted to EDS '88)

BERMUDA — AN ARCHITECTURAL PERSPECTIVE ON INTERFACING PROLOG TO A DATABASE MACHINE

Yannis E. Ioannidis [1]
Joanna Chen [2]
Mark A. Friedman [3]
Manolis M. Tsangaris

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

We describe the design and implementation of **BERMUDA**, which is a system interfacing Prolog to the Britton-Lee Intelligent Database Machine (IDM). We discuss several architectural issues faced by such systems, and we present the solutions adopted in BERMUDA. In BERMUDA, rules are stored in Prolog, and facts are primarily stored in a database. BERMUDA has been designed and implemented so that multiple concurrent Prolog processes, possibly running on different machines, can share a database. Moreover, the semantics of Prolog programs remain unchanged and the use of a database system is transparent to the user. Finally, BERMUDA has achieved a certain level of portability by using the given Prolog interpreter and database system (almost) unchanged. BERMUDA also employs several novel techniques to make the interface of Prolog to the database efficient.

[1] Partially supported by the University of Wisconsin Graduate School Research Foundation and by the National Science Foundation under Grant IRI-8703592.

[2] Author's present address, Digital Equipment Corporation, 110 Spit Brook Rd., ZKO2-3/N30, Nashua, NH 03062.

[3] Partially supported by the Resident Study Program of IBM. On leave from IBM Poughkeepsie, NY.

1. INTRODUCTION

Existing logic programming systems usually deal with knowledge bases of small size. They function based on the assumption that the knowledge base is stored in virtual memory. In addition, they provide only limited services for recovery, protection, concurrent access to distributed knowledge bases, etc., if any is provided at all. This makes them inadequate for supporting many new data/knowledge-intensive applications, such as sophisticated office automation, computer aided design and manufacturing, decision support, etc. This lack of functionality on the part of logic programming systems has motivated the development of *Deductive Database Systems*. Deductive database systems incorporate the functionality of both logic programming and database systems.

The design and construction of deductive database systems has taken many forms [Gall78, Gall81a, Dahl82, Gall84, Kowa84]. A significant role in this effort is played by the Prolog programming language [Cloc81] and systems based on it. There are four major architectures of deductive database systems:

(1) An existing logic programming system is enhanced with database functionality. Although work has been done to partially add database functionality to logic programming systems [Nais83, Scio84], doing so to an absolute level requires writing a large portion of a database system, which accounts for a considerable amount of work.

(2) An existing relational database system is enhanced with inferential capabilities. This approach is the dual of the previous one and is the one adopted in several research projects [Ston81, Nico83, Ioan84, Daya84, Ston85]. Besides inferencing, there are a few other services provided by logic programming systems that are absent from database systems, e.g. support for lists in Prolog [Park86]. The transition from a database system to a deductive database system does require some effort in enhancing the former with these new features.

(3) A system is built from scratch. This approach, when affordable, is always advantageous, because there are no previous (bad) decisions that affect the new design. The system being built at MCC [Tsur86] can be classified in this category.

(4) A logic programming system is used as a front-end to a database system. This is an obvious alternative, since logic programming systems and relational database systems are both based usually on first order logic. There are, however, significant difficulties in coupling the two together. Certain aspects of the one system have no counterpart in the other (e.g., there is a significant semantic content in the order of clauses in Prolog, which is absent from database systems). Also, they have a different processing paradigm, i.e., Prolog follows a tuple oriented algorithm, whereas database systems follow a set (relation) oriented algorithm. This mismatch makes the task of integrating the two nontrivial. This approach, however, has been tried out in many cases with noticeable success [Jark84, Chan86, Ceri86, Morr86, Bocc86].

What makes a type-4 system attractive is that it makes use of existing systems with few or no changes. Thus, the features offered by both kinds of systems, i.e., a logic programming system and a database system, are available for free. In addition, the development effort for such a system is much smaller than for a system of any other type, especially a type-3 one. Hence, although a type-3 system is expected to have superior performance than a system of any other type, a type-4 system can be functional much sooner. Efficiently interfacing the two components is the only issue that needs to be addressed in a type-4 system, since most of the remaining functionality is provided by one of the two components.

In this paper, we describe the design and implementation of **BERMUDA** (**B**rains **E**mployed for **R**ules, **M**uscles **U**sed for **D**ata **A**ccess), which is a type-4 system. BERMUDA interfaces Prolog and the Britton-Lee Intelligent Database Machine 500 (IDM) [Ubel84] under the Unix Operating system. The naming of BERMUDA was inspired by the envisioned programming environment, where rules (and possibly small sets of facts) are stored in Prolog and (large sets of) facts are stored in a database. This, in turn, was motivated by the lack of any inferencing capability on the part of a database system as opposed to Prolog, and by the lack of any sophisticated secondary storage manipulation on the part of Prolog as opposed to a database system.

The focus of this paper is on the particular architectural issues of building a type-4 system. Improvements can be made to BERMUDA by addressing several other issues that arise in all types of systems. This paper contains no discussion of such issues.

For the rest of the paper, a *database predicate* is a Prolog predicate that is stored as a relation in a database. Any other predicate is a *nondatabase predicate*. The arguments of database predicates are assumed to be either constants or variables that can never be bound, directly or indirectly, to a list, since database systems do not support lists.

The paper is organized as follows. Section 1 is an introduction. Section 2 contains a brief survey of the previous work on interfacing logic programming systems with database systems that we are aware of. It also compares that work with our work and highlights the salient aspects of our approach. In Section 3, the design of BERMUDA is given. The key aspects of the system are discussed and the various design decisions made are justified. Section 4 describes the current implementation of BERMUDA at the University of Wisconsin. Finally, Section 5 contains a summary of the paper and some directions for future work.

2. PREVIOUS WORK

Early on in the development of deductive database systems, interest was generated in interfacing Prolog as a front-end to relational database systems. There are several such systems that have already been designed and/or developed [Jark84, Chan86, Ceri86, Bocc86]. In this section, we briefly describe each one of these systems, and we identify some of their weak points that BERMUDA has overcome. For uniformity, if a system has not been given a name, we refer to it by the initials of the authors of the most significant publication describing the system.

- **JCV**

One of the first reported attempts to interface Prolog to a database system, in particular one accessed through SQL, was done at the Business School of NYU [Jark84]. The significant features of that system were the introduction of an intermediate language between Prolog and SQL, called DBCL, and the incorporation of a series of optimizations of DBCL, in addition to the query optimization of SQL done by the database system. DBCL is a variable-free subset of Prolog, similar to tableaux [Aho79]. A Prolog clause was translated into a DBCL clause, which in turn was optimized according to any relevant integrity constraints satisfied by the database. The type of constraints supported by the system were the following: value bounds (e.g., salary $\leq$ 100K), functional dependencies, and referential integrity constraints. After these optimizations, DBCL was translated into SQL in a straightforward way.

As useful and important as the above optimizations are, their application is not facilitated in any way by the fact that the system is an interface between Prolog and a database system. They could be incorporated into any general database system as well. BERMUDA, being an exercise in interfacing two such systems, does not attempt to apply such optimizations. The optimization focus of BERMUDA is on issues that arise from the particular configuration. Also, the JCV system makes two simplifying assumptions: query answers are small (and therefore caching them in an internal database in Prolog is adequate), and in any Prolog clause, all of the database predicates are grouped together. Failure of the first assumption may result in a memory overflow in Prolog. Failure of the second assumption will probably prevent the Prolog program from running. Removing the second assumption, however, is discussed as an item of future work [Jark84]. BERMUDA, on the other hand, both caches query answers externally and handles arbitrary Prolog clauses that may contain any number of sets of consecutive database predicates separated by nondatabase ones. Thus, it avoids some potential problems of the JCV system.

- **PROSQL**

PROSQL is a system developed at IBM Yorktown and involves interfacing Prolog to SQL/DS [Chan86]. In PROSQL, Prolog is enhanced with one predicate, called SQL, which takes one argument. Its argument can be any SQL statement for defining or manipulating the database. All data retrieved from a database is loaded into Prolog, which then uses them as if they were *assert*-ed in the program or loaded by *consult*-ing a file. †

A major weakness of PROSQL is that it leaves the responsibility of the interaction between Prolog and SQL/DS to the user; the use of the database system is not at all transparent. The PROSQL programmer must be familiar with both Prolog and SQL and must use the SQL predicate every time a database access is needed. Another weakness of PROSQL is that data brought from the database is simply asserted into the virtual memory of Prolog. This defeats part of the purpose of interfacing Prolog to a database system, which is to overcome the inadequacies of the virtual memory of Prolog for data-intensive applications, as the programmer is still responsible for avoiding an overflow of the memory of Prolog. The design of BERMUDA overcomes both the above problems.

† *assert* is a Prolog predicate that adds a specific clause into the program. *consult* does the same for a set of clauses written in a file.

- CGW

Another effort to interface Prolog to a database system has taken place at Stanford University [Ceri86]. This design is the one closest to the design of BERMUDA. Its basic characteristic is a loading mechanism by which, as Prolog asks for access to database predicates, query answers are asserted into the Prolog program. After that, Prolog keeps track of how much of the query answer has been consumed at any time. An important feature of the loading mechanism is that it keeps track of all the query answers that have been retrieved from the database, so that it may identify *query subsumption*, i.e., the answer of a query being a subset of the answer of another query [Fink82]. Thus, when a query comes in, the system checks whether the answer to the query can be found from results that have been asserted in Prolog already. It accesses the database only if this is impossible.

The CGW design achieves its goals by changing the Prolog interpreter, giving it opportunities to search for facts directly in secondary storage. In that respect, it is not truly an interface of Prolog to a database system; it could have been classified as a type-1 deductive database system, i.e., one that extends Prolog with database capabilities. We refer to it as a type-4 system, however, because it was claimed to be such by the designers [Ceri86], and because of some similarities it has with BERMUDA.

Query subsumption, which is one of the key aspects of the above design, is again a general optimization technique that can be incorporated into any general database system. The current version of BERMUDA does not employ such optimizations. On the other hand, the CGW design faces three problems. First, some optimization decisions are based on dictionary information that is stored in Prolog. This information is stored in the database system as well, thus introducing some redundancy in the system. Second, like PROSQL, the data is loaded into the Prolog address space, which may create an overflow for large databases. The authors do discuss overflow problems, and they do propose solutions for them, but the solutions involve more changes to Prolog [Ceri86]. Third, all the queries sent to the database are single relation queries. Joins are processed by Prolog, which may not always provide the fastest available algorithm. Each of these problems is successfully addressed by BERMUDA.

- **EDUCE**

Educe is a system developed at the European Community Research Center (ECRC) [Bocc86]. It provides an interface between Prolog and Ingres [Ston76]. In Educe, the user is able to use several new Prolog predicates that take QUEL commands in various forms as arguments. The implementation of these predicates involves some communication with INGRES, namely sending commands to it and receiving answers. The communication is done with a pair of pipes between two processes, a Prolog process enhanced with some database access methods and an INGRES process. The pipe conveniently acts as a queue for query answers, thus achieving an elegant transition from the page-at-a-time processing of INGRES to the tuple-at-a-time processing of Prolog.

Like most of the other systems above, Educe has changed Prolog significantly. This has implications at the user interface level, since transparency is lost. The user has to know both Prolog and QUEL and has the responsibility of directing commands to the database. Also, by relying on a single pair of pipes for buffering and communication between the two systems, the opportunities for caching query answers is lost. Thus, queries that are generated again because of backtracking must be reexecuted by INGRES. BERMUDA makes the database system completely transparent to the user, and by using an explicit buffering mechanism, it is able to cache and reuse query answers. Also, BERMUDA avoids several potential problems that have been attributed to systems with the same design philosophy, and which led to the design of Educe [Bocc86]. Such problems include operating systems limitations on the number of open files, the number of pipes, and the number of concurrent child processes, and also performance degradation due to uncontrolled use of system resources.

3. THE DESIGN OF BERMUDA

The current design of BERMUDA makes two assumptions:

- The order of access to tuples of database predicates is not important.
- There are no updates to database predicates, at least not during the execution of Prolog programs.

The first assumption is needed so that execution can be sped up by using different indices on database predicates for different queries. Presumably, database predicates are large, so the Prolog programmer does not know the order of the tuples and does not rely on that. The second assumption is motivated by

simplicity. Combining the semantics of Prolog with the semantics of database updates generates several difficult problems, which are not discussed here since they are outside the scope of this paper. In addition, the initial focus of the development of BERMUDA is on providing deductive capabilities over large databases. We intend to incorporate updates in a future design of BERMUDA.

Assuming the above, BERMUDA has been designed with the following goals in mind:

- Although the semantics of Prolog sometimes have undesirable effects, programming in BERMUDA should be the same as programming in Prolog. Hence, independent of how BERMUDA is implemented, the impression given to the user should be that of Prolog.
- The existence of a database system underneath Prolog should be transparent to the user. Whether something is stored in the virtual memory of Prolog or on disk should not affect any Prolog programs.
- BERMUDA should be executed as efficiently as possible. This is most important when the amount of data that Prolog has to handle is very large, where its virtual memory mechanism is inadequate to provide good performance.
- The system should allow sharing of data in the database among multiple Prolog programs.
- Both Prolog and the database system should be used unchanged, or else with minimal changes that in no way alter their basic features or the philosophy of their design.

The process structure of BERMUDA, as motivated by the above goals, is shown in Figure 3.1. Multiple Prolog processes communicate with one process called the *BERMUDA Agent.* The BERMUDA Agent in turn, communicates with a fixed number of *Loaders.* Finally, the Loaders communicate with IDM, passing queries to it and collecting the answers.

The flow of data in BERMUDA is the following. Whenever Prolog sees a sequence of database predicates, it sends the appropriate query to the BERMUDA Agent. The BERMUDA Agent formulates the appropriate SQL query and sends it to an idle Loader, if there is any, or puts it on a queue to wait until a Loader becomes free. The Loader passes the query on to IDM and collects the answer, which is stored in a Unix file. The BERMUDA Agent reads the file and starts providing Prolog with tuples at the pace at which Prolog asks for them. There can be multiple queries being manipulated by the BERMUDA Agent at the

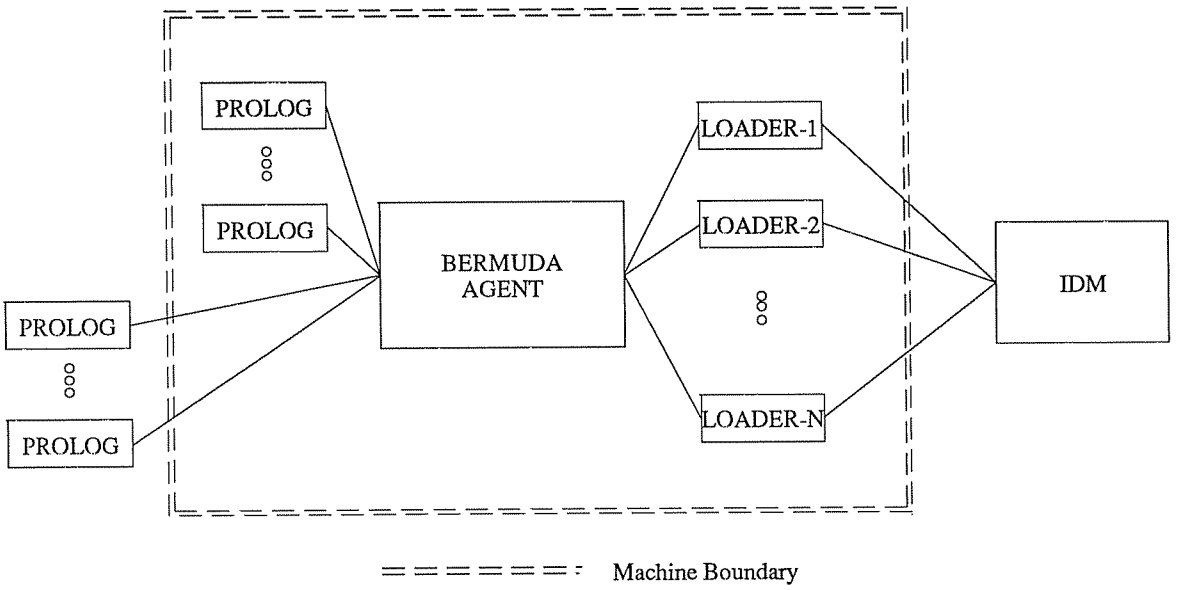

Figure 3.1: The process structure of BERMUDA.

same time, coming either from the same Prolog process or from different ones.

The most significant aspects of the design of BERMUDA are analyzed in the following subsections. These are the following: separating rules from facts, sending to the database system as few queries as possible (i.e., by transforming a sequence of database predicates into a single query), caching query answers by the BERMUDA Agent, prefetching query answers for Prolog, supporting multiple Prolog processes, employing multiple Loaders, and handling the special, extralogical constructs of Prolog, like the cut (!) symbol.

3.1. Separating Rules from Data

Motivated by the goal of keeping Prolog and IDM unchanged, the separation of rules and data was natural. IDM cannot support any form of rules (with the exceptions of view definitions and integrity constraints, which are very simplified and restricted forms of rules supported by most relational database systems). Hence, any non-ground clause of a Prolog program has to remain under the control of and be stored within Prolog. On the other hand, whenever a huge number of ground clauses is associated with a predicate, it is wise to have the predicate stored as a relation under a database system. Thus, one can take advantage of the sophisticated manipulation of secondary storage offered by the database system and

escape the generalities and performance deficiencies of Prolog's virtual memory. Another advantage of storing large predicates in the database is that this makes the predicates persistent, without needing to be reasserted every time a Prolog program is using them. Of course, the BERMUDA programmer is not constrained to put all large predicates in the database or to put all small predicates in Prolog. We believe, however, that it is mostly the large predicates that both will tend to be persistent and will improve performance when stored in a database.

3.2. Minimizing the Number of Database Queries

Crossing the boundaries of two machines, i.e., going from a Loader to IDM, is expensive. Hence, minimizing the number of times the system has to cross these boundaries is one of BERMUDA's goals. This is achieved by collecting together all the database predicates that appear consecutively in a Prolog clause and sending them as a single query to the database system. This, in turn, has the additional and even more significant benefit that the query optimizer of the database system is used. Most relational database systems [Ston76, Astr76], support several query processing algorithms, as opposed to Prolog, which supports a fixed one, and they employ sophisticated optimization techniques to choose the most efficient algorithm for each query. Therefore, assigning as much processing as possible to the database system has important benefits for overall performance. Two examples of transforming a sequence of database predicates into a single query are given below, one for a join and one for a selection. In all the forthcoming examples of Prolog programs, all database predicates are denoted by $\mathbf{d}_i$ and all nondatabase predicates are denoted by $\mathbf{p}_i$, where i is an integer. Also, for simplicity, we omit the arguments of the predicates whenever they play no role in the discussion.

Assume that some Prolog program has to evaluate the following clause:

$$\mathbf{p}_1 :- \cdots, \mathbf{p}_2, \mathbf{d}_1, \mathbf{d}_2, \mathbf{p}_3, \cdots.$$

If "$\mathbf{d}_1,\mathbf{d}_2$" involves a join between $\mathbf{d}_1$ and $\mathbf{d}_2$, BERMUDA sends "$\mathbf{d}_1,\mathbf{d}_2$" as a single query to the database system, as opposed to sending two separate queries "$\mathbf{d}_1$" and "$\mathbf{d}_2$". IDM has the capability to both consider the use of more join algorithms than Prolog, e.g., merge scan [Seli79], and to take advantage of any possible indices on $\mathbf{d}_1$ and/or $\mathbf{d}_2$. On the other hand, if "$\mathbf{d}_1,\mathbf{d}_2$" does not involve a join between $\mathbf{d}_1$ and $\mathbf{d}_2$ (i.e., it represents a cross product of the two predicates), BERMUDA sends two separate queries, "$\mathbf{d}_1$" and "$\mathbf{d}_2$".

Similarly, assume that a Prolog program has to evaluate the following clause:

$$\mathbf{p}_1 :- \cdots, \mathbf{p}_2, \mathbf{d}_1(...,X,...), X>10, \mathbf{p}_3, \cdots.$$

BERMUDA will again choose to send "$\mathbf{d}_1(...,X,...),X>10$" as a single query to the database system. Any indices existing on the column of $\mathbf{d}_1$ where X appears can thus be used to apply the selection "*X>10*" and minimize processing time.

Clearly, the more database predicates are collected together and transformed into a single query, the more performance will benefit from the query processing and optimization techniques of the database system. With this goal in mind, the question arises as to whether or not it is possible to make a single query out of a set of database predicates, even if they do not appear consecutively in a Prolog clause. As an example, consider the following clause:

$$\mathbf{p}_1 :- \cdots, \mathbf{p}_2, \mathbf{d}_1, \mathbf{p}_3, \mathbf{d}_2, \mathbf{p}_4, \cdots.$$

It would be nice if we could still group together "$\mathbf{d}_1,\mathbf{d}_2$" as a single query to the database system. The semantics of Prolog, however, have to be retained, since this was one of our original goals. More specificly, this means that, the original Prolog query should have the same set of answers (even if individual answers appear in a different order), and all the external files should have the same contents. The problem can also be formulated as a question of whether two consecutive predicates in a Prolog clause can be swapped (i.e., changing the order of access) yielding an equivalent program (i.e., retaining the program semantics) or not. There are several cases where swapping does preserve the semantics of a Prolog program, and can thus be safely applied to group more database predicates into fewer database queries. The current design and implementation of BERMUDA, however, do not support such swapping of predicates.

3.3. Caching Query Answers

In trying to minimize the number of queries sent to the database system, BERMUDA has to face another problem besides the one mentioned in the previous section: due to the backtracking of Prolog, the same query may be sent to the BERMUDA Agent multiple times. For example, consider the following Prolog program:

$\mathbf{p}_1(X,Y) :- \mathbf{p}_2(X,Z), \mathbf{d}_1(Z,Y).$

$\mathbf{p}_2(2,1).$

$\mathbf{p}_2(3,1).$

$\mathbf{p}_1(X,Y)$?

Following the execution of Prolog, the query "$\mathbf{d}_1(1,Y)$" will be generated after $\mathbf{p}_2$ is resolved with its first ground clause, and the same query will be generated again after $\mathbf{p}_2$ is resolved with its second ground clause. To avoid sending the same query to the database system again, the BERMUDA Agent caches query answers. These answers are stored in flat files that are accessible to the BERMUDA Agent. The BERMUDA Agent keeps track of the queries that have been executed in IDM and where their answers reside. When a duplicate query is sent to the BERMUDA Agent, the answer is ready to be retrieved from the file and sent back to the requesting Prolog process. Garbage collection is applied to remove old query answers.

3.4. Prefetching Query Answers

Due to the differences in speed and processing model of the various modules of BERMUDA, attention has to be paid to their correct synchronization so that they are not unnecessarily blocked waiting for responses from other modules. This is especially crucial at the interface between Prolog and the BERMUDA Agent, since Prolog accesses virtual memory, whereas the BERMUDA Agent retrieves query answers from disk. The speed difference between virtual memory and disk may make the BERMUDA Agent a potential performance bottleneck, forcing Prolog processes to block. As an example, consider the following Prolog program:

$\mathbf{p}_1(X,Y) :- \mathbf{d}_1(X,Z),\mathbf{p}_2(Z,Y).$

$\mathbf{p}_2(2,1).$

$\mathbf{p}_2(3,1).$

$\mathbf{p}_1(X,Y)$?

Assuming that $\mathbf{d}_1$ is a large predicate, the file containing the answer to the query "$\mathbf{d}_1(X,Y)$", i.e., the contents of $\mathbf{d}_1$, will span several pages. After all the tuples of one page have been sent to and examined by Prolog, the next page has to be retrieved. Unless some provision is taken, Prolog will have to block waiting for an I/O to happen. To avoid this, the BERMUDA Agent uses prefetching. As Prolog reaches a tuple in a query answer that is close to the end of the current page, the BERMUDA Agent asks for the next page of the query answer to be brought into its buffers. Thus, the I/O happens while Prolog manipulates the remaining tuples of the previous page. Then, when the first tuple in the new page is requested, it is already

in the buffer of the BERMUDA Agent, and Prolog does not have to block.

Ideally, the timing of prefetching (i.e., when prefetching actually happens) should depend on the specifics of the Prolog program. In particular, the more work the Prolog program has to do between two consecutive requests for tuples from a database query answer, the more the prefetching should be delayed so that valuable buffers are not unnecessarily occupied.

3.5. Supporting Multiple Prolog Processes

As described earlier and shown in Figure 3.1, BERMUDA supports multiple Prolog processes running concurrently, accessing the same database. The Prolog processes may run on the same machine as the BERMUDA Agent or on different machines. Since all database queries are sent to the database by the BERMUDA Agent, and not directly by the Prolog processes issuing them, the BERMUDA Agent should protect the database from any unauthorized access. Not having database protection as a goal (see beginning of Section 3), the current design of BERMUDA does not check for user authorization. This is intended to be part of a future version of the system.

3.6. Employing Multiple Loaders

The decision to use IDM as our database system was made primarily for performance and availability reasons. The unfortunate aspect of this decision was that there is no asynchronous interface to IDM. Whenever a query is sent to IDM, the sender has to block and wait for the complete answer to be prepared before running again and collecting it. Interfacing the BERMUDA Agent directly to IDM would thus require the BERMUDA Agent to block every time a query was sent to the database system. This, in turn, would cause all the Prolog processes with database queries to wait until the BERMUDA Agent came back again to accept the next query. This would effectively serialize the database accesses of the Prolog processes, providing no concurrency.

To avoid the above problem, we have placed Loader processes between the BERMUDA Agent and IDM. Their sole purpose is to free the BERMUDA Agent from blocking every time a query is sent to IDM. Instead, the query is sent to a Loader, and it is the Loader that blocks when the query is passed to IDM. At the same time, the BERMUDA Agent remains available for both accepting more queries from and sending data back to other Prolog processes. When the Loader starts collecting the query answer, it

notifies the BERMUDA Agent, which in turn starts feeding the appropriate Prolog process with tuples from the answer. This creates a pipeline between Prolog and IDM, with Prolog manipulating the first tuples from the query answer before the complete answer is formed. With respect to the Loader configuration, two questions had to be answered: Should there be multiple Loaders, or only one? And, should a Loader be created and destroyed for each database query, or should Loaders live for the duration of the program? Our decision was that there should be multiple Loaders that live for the duration of the program, and it is justified in the next two paragraphs.

Using only one Loader process is not enough to avoid the blocking problem completely. Blocking of the BERMUDA Agent is avoided, but the problem arises again if another database query is sent to the BERMUDA Agent before the answer to the first one is collected. The Loader process is blocked, so the BERMUDA Agent has to wait for it to unblock before sending the new query. Moreover, with one Loader, only one query can be sent to IDM at any one time, thus taking no advantage of IDM's ability to concurrently handle multiple users and multiple queries. For this reason, we have chosen to have multiple Loader processes in BERMUDA. Each one handles at most one query at a time.

Creating a new Loader process for every query and destroying the process at the end is an expensive operation. This becomes significant if we take into account that a Prolog program, due to backtracking, may issue a huge number of database queries. Therefore, BERMUDA has a fixed number of Loaders that are created at system invocation time and remain alive until the session is ended. With this scheme, it is conceivable that at some point there will be more queries issued to the BERMUDA Agent than there are Loaders. These queries are put in a priority queue in the BERMUDA Agent and are served by the Loaders using a First-Come-First-Served policy. If a query is queued, it will be serviced by the first Loader that becomes available after the query reaches the front of the queue.

Notice that a single Prolog process may have multiple database queries active at the same time. This situation can be created by two types of clauses: recursive clauses and clauses with multiple clusters of database predicates, each separated from the next by at least one nondatabase predicate. We will give an example of the second type of clauses; recursive clauses behave similarly. Consider a Prolog process running a program containing the following clause:

$$p_1 :- \cdots, p_2, d_1, p_3, d_2, p_4, \cdots.$$

The Prolog process will first issue "d_1" as a database query (possibly with some of the variables of d_1 instantiated). As soon as it gets its first d_1 tuple back, and after it evaluates p_3, it will issue another query, "d_2". This may happen while the answer to the first query "d_1" is still being formed, thus leading to two simultaneously active queries issued by the same Prolog process. Hence, the ability to queue up database queries is essential even if we disallow multiple Prolog processes accessing the database concurrently.

3.7. Handling Extralogical Constructs of Prolog

In addition to "data" predicates, Prolog supports various extralogical predicates that are used to control the flow of data in a program. The most significant of those extralogical constructs of Prolog is the cut predicate (!). The appearance of ! in a Prolog program presents an important optimization challenge to BERMUDA. The first instantiation of the variables that makes the partial clause before the ! evaluate to "true" is the only one needed. When this happens, part of the Prolog stack is thrown away, and the program never backtracks before the !. Hence, if there are database predicates before the !, it is highly probable that a large portion of the corresponding query answers will never be used. To minimize unnecessary extra work, Prolog processes send !'s to the BERMUDA Agent. The BERMUDA Agent, in turn, notifies the appropriate Loader to stop collecting further answers if the complete answer has not already been formed, as they are not going to be used.

4. THE IMPLEMENTATION OF BERMUDA

BERMUDA has been implemented at the University of Wisconsin. The BERMUDA Agent and the Loaders together with any Prolog application run on machines supporting Unix and communicate among themselves and with IDM over a 10mb/sec local area network. Following a client/server model, communication between the BERMUDA Agent (the server) and any other process (the client) is completely asynchronous; the BERMUDA Agent never blocks to wait for another process. As we mentioned before, this does not apply to the communication between the Loaders and IDM, since no asynchronous interface to IDM exist. The Loaders have to block whenever they send a query to IDM. Otherwise, the Loaders would not be necessary.

There are three aspects of the design of BERMUDA described above that are not included in the current implementation:

- No garbage collection is applied for cached query answers.
- The timing of prefetching does not depend on the Prolog program. It currently depends only on the value of a system parameter that specifies when the next page should be prefetched in terms of the number of remaining tuples in the current page.
- In the case of a !, the BERMUDA Agent does not interrupt the collection of the answers of database queries that appear before the !.

In the future, all of these features will be included in the system.

In the implementation of BERMUDA, several decisions were forced on us by the computational environment we were using. Peculiarities of Unix and IDM were the main causes of such forced decisions. These are discussed in the following subsections, where we give the implementation details of the various parts of the system.

4.1. Prolog

For the implementation of BERMUDA we used C-Prolog, since it was the only one available to us. This immediately created a major problem: C-Prolog cannot communicate with other processes. This has been identified as a problem in other similar projects as well [Bocc86, Ghos87]. We took the straightforward approach to solving the problem, minimally betraying the fifth design goal of Section 3. We enhanced C-Prolog so that it can communicate with other processes running on the same or different machines. Communication is achieved using Unix domain sockets for processes on the same machine and TCP sockets for processes on different machines. This enhancement involves a slight change to the Prolog predicates *see* and *tell*, so that they can recognize a socket as a valid I/O port, but it does not affect any previously running Prolog program.

With the exception of some necessary declarations in the beginning, Prolog programs written for BERMUDA do not explicitly refer to the underlying database system. BERMUDA takes care of the necessary database accesses. This is achieved by having every incoming Prolog program pass through a *Preprocessor*, before it is sent to the Prolog interpreter. This is shown in Figure 4.1.

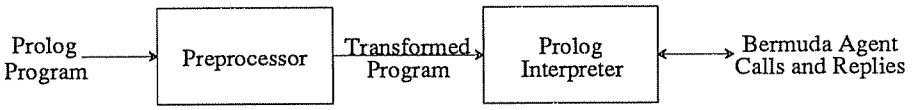

Figure 4.1: Preprocessing a Prolog program.

The preprocessor is written in Prolog, and its sole purpose is to transform the original Prolog program at *consult*-ing time by inserting the appropriate calls to the BERMUDA Agent into it. The preprocessor achieves this by performing the following tasks:

(1) identifying database predicates and replacing them with a pseudo definition that just calls the proper database access routine,

(2) marking database predicates that appear before any existing !,

(3) detecting joins and selections in sets of consecutive database predicates and encoding them into messages for the BERMUDA Agent.

The transformed program is then processed by the Prolog interpreter.

Prolog processes a given program in a top-down, tuple-at-a-time fashion, employing unification and backtracking. Whenever a set of consecutive database predicates (possibly with attached selections) is reached, a scan is opened and sent to the BERMUDA Agent to collect the answer. When the answer has been (at least partially) collected, the BERMUDA Agent notifies the appropriate Prolog process, which starts asking for tuples, one at a time, following the canonical Prolog algorithm. The responsibility for caching query answers and prefetching tuples from the disk lies with the BERMUDA Agent.

4.2. The BERMUDA Agent

The BERMUDA Agent is the heart of the entire system. It performs the following tasks:

- It directs queries to Loaders, queueing them up when all of them are busy.
- It collects query answers a page at a time from IDM (via the Loaders), passing them a tuple at a time to the Prolog processes.
- It caches query answers so that when a query is generated for a second time it is not sent to IDM again.

- It prefetches pages of query answers from disk to a buffer pool so that Prolog processes do not have to block waiting for tuples, and it manages the buffer pool.

The BERMUDA Agent in the current implementation together with its data structures is shown in Figure 4.2.

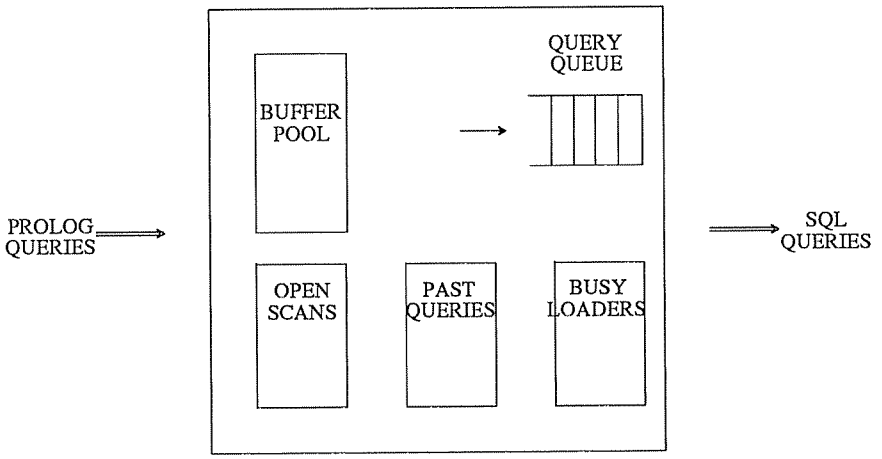

Figure 4.2: Data structures of the BERMUDA Agent.

In the following subsections we elaborate on how the BERMUDA Agent achieves the functionality we described above.

4.2.1. Connecting Prolog and IDM

The BERMUDA Agent primarily serves the role of "postman" between IDM and Prolog. It collects query answers from IDM and then passes individual tuples to the requesting Prolog process at Prolog's pace.

Query answers are communicated between the Loaders and the BERMUDA Agent through Unix files. This is somewhat unfortunate, because the Loader has to write into the file, and the BERMUDA Agent has to read from the file into its own buffer pool to send data back to Prolog. This solution, however, is forced on the implementation, because Unix does not support shared memory between processes. If this were not the case, the Loaders and the BERMUDA Agent would communicate via shared memory.

When a Prolog process issues a database query to the BERMUDA Agent, the latter makes an entry into the *Open Scans* data structure, where it keeps information about the query and the originating Prolog process. The BERMUDA Agent then looks into the *Past Queries* structure to see if the query has been

answered before. If it has, the appropriate file is opened, and tuples are retrieved to be passed to Prolog (unless the first page of the file is already in the buffer pool, in which case tuples may be immediately returned to Prolog). If the query has not been answered before, but it appears in the query queue already or is currently being processed by a Loader, a note of this fact is made so that the query is not sent to IDM again. Otherwise, the query is put at the end of the query queue to wait for a Loader to become available. Information about which Loaders are busy with which query is stored in the *Busy Loaders* structure. When the Loader has collected a page's worth of the answer, it stores it into a Unix file and then notifies the BERMUDA Agent, which in turn takes the first tuple and passes it to Prolog. After that, whenever Prolog asks for a tuple, the BERMUDA Agent passes it on. While this is happening, the Loader may still be appending more of the query answer to the Unix file. When the complete answer has been written out, the Loader notifies the BERMUDA Agent that it has become available and is removed from *Busy Loaders*.

4.2.2. Queuing up Queries for the Loaders

As we mentioned above, whenever a query is given to the BERMUDA Agent and all of the Loaders are busy, the BERMUDA Agent puts the query at the end of a queue. The queue is served on a First-Come-First-Served basis and is shared by all of the Loaders. Our choice of this scheduling policy has been motivated by simplicity. We also expect that it is the optimal and most fair policy in most cases. There are some cases, however, where a more sophisticated policy is more appropriate. For example, consider the following clause:

$$\mathbf{p}_1 :- \cdots, \mathbf{p}_2, \mathbf{d}_1, \mathbf{p}_3, \mathbf{d}_2, \mathbf{p}_4, \cdots.$$

This will generate two queries, namely "$\mathbf{d}_1$" and "$\mathbf{d}_2$". It is conceivable that at some point the answer for "$\mathbf{d}_1$" is partially computed, a $\mathbf{d}_1$ tuple has been passed to Prolog, the query "$\mathbf{d}_2$" has been sent, and all of the Loaders are busy. In this case, it makes more sense for "$\mathbf{d}_2$" to preempt the processing of "$\mathbf{d}_1$" and take over its Loader, as all of its tuples will be requested by Prolog before any other tuple of "$\mathbf{d}_1$" is requested. In future versions of BERMUDA, we plan to include more sophisticated scheduling policies that take into account this sort of available knowledge about the system's behavior.

4.2.3. Managing the Buffer Pool

The BERMUDA Agent maintains a buffer pool with pages of various query answers that are requested from Prolog processes. Scans from *Open Scans* point into these buffers, keeping track of the next tuple to be sent to Prolog. Several scans may point to the same buffer. As described earlier, the BERMUDA Agent prefetches pages from a query answer so that Prolog does not have to wait for I/O. Due to the particular semantics of Prolog clauses and the processing algorithm Prolog uses, the page replacement policy used for the buffer pool is somewhat unconventional. We define the *least recently used query* to be the query with the least recently used page in the buffer pool. Then, when a page needs to be brought in and all of the pages in the buffer pool are occupied, the page chosen to be replaced is determined as follows:

(1) if there is at least one page in the buffer pool that has no active scans referencing it, then the most recently used such page of the least recently used query is chosen (this is motivated by the nested-loops referencing pattern of Prolog); otherwise,

(2) if there is at least one prefetched page in the buffer pool, i.e., one for which no Prolog has requested any tuples from it yet (but it probably will in the future), then the prefetched page of the least recently used query is chosen; otherwise,

(3) (ii) the (sole) page belonging to the least recently used query is chosen.

The above algorithm takes into account the semantics of Prolog to guarantees that, for queries that come from the same Prolog clause, there will never be a replacement of a page of a query by a page of another query to the right of the first query.

As an example of the replacement algorithm, consider evaluating the following clause:

$$\mathbf{p}_1 :- \cdots, \mathbf{p}_2, \mathbf{d}_1, \mathbf{p}_3, \mathbf{d}_2, \mathbf{p}_4, \mathbf{d}_3, \cdots.$$

Suppose that Prolog has requested several answers from the database queries "$\mathbf{d}_1$", "$\mathbf{d}_2$", and "$\mathbf{d}_3$", and the BERMUDA Agent has filled the buffer pool with the first 3 pages of the file of "$\mathbf{d}_1$", the first 2 pages of the file of "$\mathbf{d}_2$", and the first four pages of the file of "$\mathbf{d}_3$". Furthermore, suppose that the BERMUDA Agent has prefetched page 4 for "$\mathbf{d}_1$" and page 3 for "$\mathbf{d}_2$". Then, the buffer pool looks as follows ($\mathbf{d}_i$.j means page j from query "$\mathbf{d}_i$"):

Query	Resident Pages in Most Recently Used Order
"$\mathbf{d}_1$"	$\mathbf{d}_1$.3 (active), $\mathbf{d}_1$.4 (prefetched), $\mathbf{d}_1$.2, $\mathbf{d}_1$.1
"$\mathbf{d}_2$"	$\mathbf{d}_2$.2 (active), $\mathbf{d}_2$.3 (prefetched), $\mathbf{d}_2$.1
"$\mathbf{d}_3$"	$\mathbf{d}_3$.4 (active), $\mathbf{d}_3$.3, $\mathbf{d}_3$.2, $\mathbf{d}_3$.1

Table 4.1. Buffer pool contents.

Assume that, due to Prolog backtracking on $\mathbf{d}_3$, the BERMUDA Agent prefetches $\mathbf{d}_3$.5. Then, according to step (1), $\mathbf{d}_1$.2 is selected for replacement. If several new active pages were brought into the buffer pool, due to additional database predicates to the right of $\mathbf{d}_3$, the order of selection of pages for replacement would be $\mathbf{d}_1$.1, $\mathbf{d}_2$.1, $\mathbf{d}_3$.3, $\mathbf{d}_3$.2, $\mathbf{d}_3$.1, $\mathbf{d}_1$.4, $\mathbf{d}_2$.3, $\mathbf{d}_1$.3, $\mathbf{d}_2$.2, etc.

4.3. The Loaders

As we have described earlier, the purpose of the Loaders is to cope with the lack of an asynchronous interface to IDM, and to take advantage of the capabilities of IDM to handle multiple concurrent users. An idle Loader receives an SQL query [Astr76] from the BERMUDA Agent, sends it to IDM, and then blocks. When IDM is ready with the answer, it notifies the Loader, which wakes up and starts collecting it. The answer can be given out by IDM only one tuple at a time. To speed up the service of other parts of the system, the Loader continuously asks for tuples until it fills a page. It then writes the page in a specified file, notifies the BERMUDA Agent that the answer is partially ready and tuples can be sent to the appropriate Prolog, and turns back to IDM to continue collecting the query answer. When the complete query has been written into the file, it notifies the BERMUDA Agent and receives the next query for processing (if there is one waiting in the queue).

5. SUMMARY AND FUTURE WORK

We have described the design and implementation of BERMUDA, which is a system interfacing Prolog to the Britton-Lee Intelligent Database Machine (IDM). BERMUDA introduces the BERMUDA Agent and a set of Loaders between possibly several Prolog processes and IDM. The role of the BERMUDA Agent is to receive database queries from Prolog, send them via the Loaders to IDM, and after the answer is collected, send the tuples of the answer back to Prolog one at a time. In addition, the BERMUDA Agent caches query answers for future use, prefetches query answers from disk to avoid delaying Prolog, manipulates a common buffer pool storing query answers requested by the various Prolog

processes, and manipulates the queue of queries waiting to be serviced by the Loaders. The role of the Loaders is to directly communicate with IDM, sending queries and receiving answers. They are introduced to free the BERMUDA Agent from having to block for every query sent to IDM, since the latter lacks an asynchronous interface.

In addition to several small-scale enhancements and improvements of the design and implementation of BERMUDA, we are currently working on two major aspects of the system. First, we are investigating the effects to the semantics of a Prolog clause caused by changing the position of predicates in the clause. We are mostly interested in identifying cases where database predicates can "move over" nondatabase predicates without affecting the semantics of the clause. This should allow more database predicates to be grouped together in a single database query (see Section 3.2). Second, we are now comparing the performance of BERMUDA with the performance of Prolog and the performance of IDM. We use the Wisconsin benchmark [Bitt83] as the core of our experiments, enhanced with some additional queries that involve nondatabase predicates as well. We hope that the results of these experiments will shed some light on the validity of the design and implementation decisions of BERMUDA, and also on the effectiveness of type-4 deductive database systems in general.

Acknowledgments: We would like to thank Mike Carey and Timos Sellis for valuable comments on earlier drafts of the paper.

6. REFERENCES

[Aho79]
Aho, A., Y. Sagiv, and J. Ullman, "Equivalences Among Relational Expressions", *SIAM Computing Journal* **8**, 2 (May 1979), pages 218-246.

[Astr76]
Astrahan, M. et al., "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems* **1**, 2 (June 1976), pages 97-137.

[Bitt83]
Bitton, D., D. J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", in *Proc. 9th International VLDB Conference* , Florence, Italy, August 1983, pages 8-19.

[Bocc86]
Bocca, J., "EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS", in *Proc. of the 1986 Symposium on Logic Programming*, Salt Lake City, UT, September 1986, pages 36-45.

[Ceri86]
Ceri, S., G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently", in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, pages 141-153.

[Chan86]
Chang, C. L. and A. Walker, "PROSQL: A Prolog Programming Interface with SQL/DS", in *Expert Database Systems, Proceedings from the First International Workshop*, edited by L. Kerschberg, Benjamin/Cummings, Inc., Menlo Park, CA, 1986, pages 233-246.

[Cloc81]
Clocksin, W. F. and C. S. Mellish, *Programming in Prolog*, Springer Verlag, New York, N.Y., 1981.

[Dahl82]
Dahl, V., "On Database Systems Development through Logic", *ACM TODS* 7, 1 (March 1982), pages 102-123.

[Daya84]
Dayal, U. et al., *"Knowledge-Oriented Database Management"*, Technical Report, CCA-84-02, Computer Corporation of America, Cambridge, MA, 1984.

[Fink82]
Finkelstein, S., "Common Expression Analysis in Database Applications", in *Proceedings of the 1982 ACM-SIGMOD Conference on the Management of Data*, Orlando, FL, June 1982, pages 235-245.

[Gall78]
Gallaire, H. and J. Minker, *Logic and Data Bases*, Plenum Press, New York, N.Y., 1978.

[Gall81a]
Gallaire, H., "Impacts of Logic on Data Bases", *Proc. 7th International VLDB Conference*, Cannes, France, August 1981, pages 248-259.

[Gall84]
Gallaire, H., J. Minker, and J. M. Nicolas, "Logic and Databases: A Deductive Approach", *ACM Computing Surveys* **16**, 2 (June 1984), pages 153-185.

[Ghos87]
Ghosh, S., C. C. Lin, and T. Sellis, *Implementation of a Prolog-INGRES Interface*, Unpublished Manuscript, University of Maryland, College Park, MD, September 1987.

[Ioan84]
Ioannidis, Y. E., L. D. Shinkle, and E. Wong, "Enhancing INGRES with Deductive Power" (Position Paper), in *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Isl., SC, October 1984, pages 847-850.

[Jark84]
Jarke, M., J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System", in *Proceedings of the 1984 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1984, pages 296-306.

[Kowa84]
Kowalski, R. A., "Logic as a Database Language", in *Proc. 3rd British National Conference on Databases*, edited by J. Longstaff, Leeds, U.K., July84, pages 103-132.

[Morr86]
Morris, K., J. D. Ullman, and A. VanGelder, "NAIL! System Design Overview", in *Proc. of the 3rd International Conference on Logic Programming*, London, England, July 1986, pages 554-568.

[Nais83]
Naish, L. and J. A. Thom, *"The MU-Prolog Deductive Database"*, Technical Report 83/10, Computer Science Dept., University of Melburne, November 1983.

[Nico83]
Nicolas, J. M. and K. Yazdanian, "An Outline of BDGEN: A Deductive DBMS", in *Information Processing 83*, edited by R. E. Mason, North Holland, 1983, pages 711-717.

[Park86]
Parker, R. et al., "Logic Programming and Databases", in *Expert Database Systems, Proceedings from the First International Workshop*, edited by L. Kerschberg, Benjamin/Cummings, Inc., Menlo Park, CA, 1986, pages 35-48.

[Scio84]
Sciore, E. and D. Warren, "Towards an Integrated Database-Prolog System", in *Proceedings of the 1st International Workshop on Expert Database Systems*, Kiawah Isl., SC, October 1984, pages 801-815.

[Seli79]
Selinger, P. et al., "Access Path Selection in a Relational Data Base System", in *Proceedings of the 1979 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1979, pages 23-34.

[Ston76]
Stonebraker, M., E. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems* **1**, 3 (September 1976), pages 189-222.

[Ston81]
Stonebraker, M., R. Johnson, and S. Rosenberg, *"Extending INGRES with a Rules System"*, Memorandum No. UCB/ERL M81/93, University of California, Berkeley, December 1981.

[Ston85]
Stonebraker, M., "Triggers and Inference in Data Base Systems", in *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, February 1985.

[Tsur86]
Tsur, S. and C. Zaniolo, "LDL: A Logic-Based Data-Language", *Proc. 12th International VLDB Conference*, Kyoto, Japan, August 1986, pages 33-41.

[Ubel84]
Ubell, M., "The Intelligent Database Machine (IDM)", in *Query Processing in Database Systems*, edited by W. Kim, D. Reiner, and D. Batory, Springer-Verlag, New York, N.Y., 1984.